1. **Monolithic Architecture :**

A Monolithic Architecture is a single-tiered software application where all components are tightly coupled and run as one single unit.

It’s like building one big application that handles everything: UI, business logic, database access — all bundled together. Presentation Layer (UI) , Business Logic Layer, Data Access Layer , Database All layers are part of one codebase and deployed together.

**Real-World Example:**

Imagine an e-commerce website built monolithically:

User Login

Product Catalogue

Cart

Payment

Notifications

All these modules are in the same application.

When you deploy — you deploy the entire thing together.

**Advantages of Monolithic Architecture**

1. Simple to develop - One project, one build process
2. Easy to deploy - Only one deployable file (e.g., WAR, JAR, EXE)
3. Easy to debug - Everything in one place — stack traces are simpler
4. Performance - No network calls between modules

**Disadvantages**

1. Hard to scale - You scale the whole app, not just one module
2. Slower development - Large codebase = difficult collaboration
3. Long deployment times - Even small changes need full redeploy
4. Not modular - One bug can crash the whole app
5. Hard to adopt new tech - You’re locked into one language/stack

When to Use Monolithic?

1. Small to medium-sized applications
2. When team size is small
3. Fast initial development
4. Simple deployment requirements

**Microservices Architecture :**

**(REST API, Feign Client, Eureka Server, API Gateway, Circuit Breaker)**

Microservices is an architectural approach where a large application is built as a collection of small, independent services, each responsible for a specific functionality.

Each microservice:

1. Has its own codebase
2. Can use its own database
3. Communicates with other services via APIs

**Real-World Example: E-Commerce**

Microservice Responsibility

User Service - Login, registration

Product Service - Product catalogue, inventory

Cart Service - Manage shopping cart

Payment Service - Handle payments

Notification Service - Send email/SMS notifications

Each run on its own server, can be written in different languages, and scaled separately.

**Benefits of Microservices**

Advantage Explanation

**Fault Isolation -**  One failure doesn’t crash entire system

**Independent Deployment -** No need to redeploy the whole app, Small, manageable services

**Scalability -** Scale only what’s needed (e.g. payments), Easier scaling

**Tech Flexibility** **-**  Use different languages/frameworks, Tech freedom

**Team Autonomy -**  Teams can own specific services

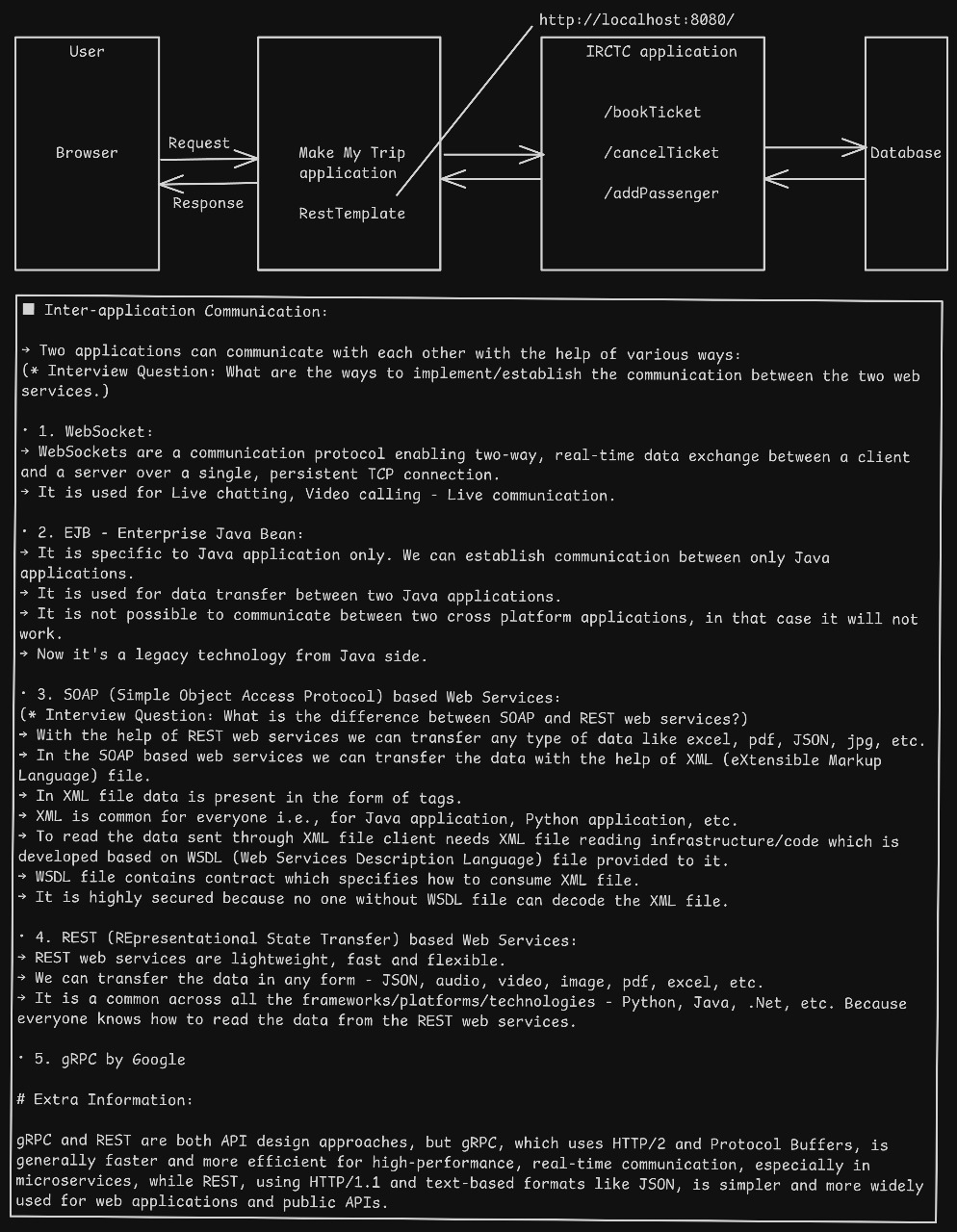
Microservices = Divide & Conquer for modern apps

Needs good planning, and observability

**Disadvantages:**

Microservices can be harder to manage because there are many small services to handle, they need reliable communication between them, keeping data consistent is tricky, and it requires more effort for testing, monitoring, and deployment.

**How Services Communicate**



**Synchronous Communication –** REST,SOAP, EJB, FeignClient , RestTemplate, WebClient

**Synchronous Communication**

In synchronous communication, the producer service sends a request and waits for the response before continuing, which introduces blocking and potentially high latency. This model is suitable **when real-time feedback is essential**. Common tools supporting this pattern include REST, SOAP, EJB, Spring’s FeignClient, and WebClient.

**1. REST API (Representational State Transfer)**

1. **REST** is used for synchronous communication that communicates over the **HTTP** protocol.

and uses standard HTTP methods like **GET**, **POST**, **PUT**, and **DELETE** to perform actions.

1. REST is **stateless**, meaning the server **doesn’t remember past requests**. It commonly uses **JSON** to send data, but can also use **XML, text, or files**.
2. REST APIs are **easy to use, scalable**, and work with **any programming language**.
3. They are widely used in web and mobile apps, and tools like **Postman** and **Swagger** help with testing and documentation.

**2. SOAP (Simple Object Access Protocol)**

1. **SOAP** is a **synchronous** communication used in **enterprise-level applications** like banking, finance, and insurance, where **security and reliability** are critical.
2. It works over protocols like **HTTP** and **SMTP**, and strictly uses **structured XML format** for messaging which is not easily readable without knowing its structure. As a result, it is **highly secure**.
3. To interpret this structure, the **producer** provides a **WSDL (Web Services Description Language)** file, containing all the details about the service — including the methods, data types, and message structure.
4. Only the **consumer** with access to the WSDL file can understand and parse the XML, making it secure by design.
5. The consumer uses the WSDL to automatically generate structure needed to send and receive SOAP messages correctly.
6. When a SOAP call is made, the **receiver** gets the message along with the **WSDL-based instructions**, ensuring that only authorized systems can understand the content. However, reading and processing XML requires **more infrastructure** and is more **resource-heavy than formats like JSON**.
7. SOAP also includes **built-in error handling, message integrity, and WS-Security** features like encryption and authentication, making it a preferred choice for systems that demand strict standards and robust communication.

**3. FeignClient (Spring Cloud)**



**E.g. Book a ticket in MMT using IRCTC service**

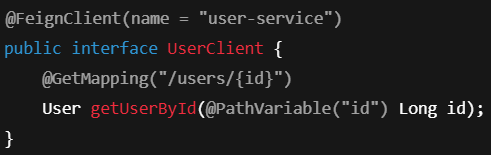
**Feign** is a **declarative HTTP client** — it allows you to write HTTP clients interface with abstract methods and annotating them. No need to manually write REST template or WebClient logic.

Spring Cloud integrates Feign with @FeignClient, making it easy to consume REST APIs from other microservices. Spring Boot + Spring Cloud will auto-generate the implementation for you at runtime.

Feign --> Spring Cloud LoadBalancer --> Eureka

**Key Features**:

1. **Declarative REST client** **using interfaces**
2. Integrates with Eureka (service discovery), Ribbon (load balancing)
3. Reduces boilerplate code



**4. RestTemplate (Spring Framework) (Deprecated – do not tell about this)**

1. **Type**: Synchronous (blocking)
2. **Protocol**: HTTP
3. **Format**: JSON, XML
4. **Used For**: REST API calls in legacy Spring apps
5. **Key Features**:
6. Simple to use
7. Now deprecated in favor of WebClient

**5. WebClient :**

WebClient is a **non-blocking, reactive HTTP client** introduced in **Spring 5** as part of the **Spring WebFlux** module. It is used to **make HTTP calls to external services**.

* **Asynchronous & non-blocking** — ideal for reactive applications.
* Replaces the older **RestTemplate** for new development.
* Handles **GET, POST, PUT, DELETE**, etc., with ease.

**6. EJB (Enterprise Java Beans) Java-Specific Communication**

1. **Type**: Synchronous
2. **Protocol**: Java RMI (Remote Method Invocation)
3. **Used For**: Legacy enterprise applications (Java EE)
4. **Key Features**:
5. Supports transaction management and security
6. Runs inside an application server
7. Considered heavy and outdated in modern architectures

**Asynchronous Communication**

1. **Definition:** Caller does not wait; message is queued or event is triggered.
2. **Latency:** Lower for caller; depends on consumer.
3. **Examples:** Kafka, RabbitMQ, WebClient (non-blocking), WebSocket
4. **Use Case:** Log processing, event sourcing, chat systems

**1. RabbitMQ (Message Broker)**

1. **Type**: Asynchronous (Message Queue)
2. **Protocol**: AMQP
3. **Format**: Any (JSON, XML, etc.)
4. **Used For**: Background jobs, communication between decoupled services
5. **Key Features**:
6. Reliable message delivery
7. Acknowledgement and retry support
8. Good for small to medium-scale systems

So, currently we are using REST API calls / REST template but we are planning to remove REST template and replace it with RabbitMQ. WHY? Because when producer produces the response then that response is necessary to reach to the consumer but when consumer service is down for whatever the reason is it couldn’t reach to the consumer. But with the help of RabbitMQ this won’t happen as RabbitMQ stores the response produced by the producer when consumer service is down and whenever the consumer service is up then that response is received and read by the consumer service and there will be **no message failure**. RabbitMQ stores the responses in the form of queues

If you want that msg should receive consumer whether it takes time then we use RabbitMQ. **It is slow.**

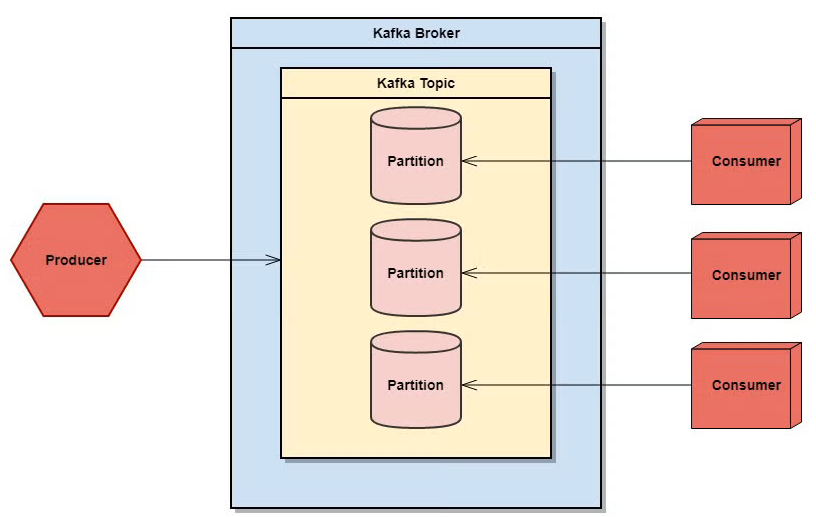
**2. Apache Kafka**

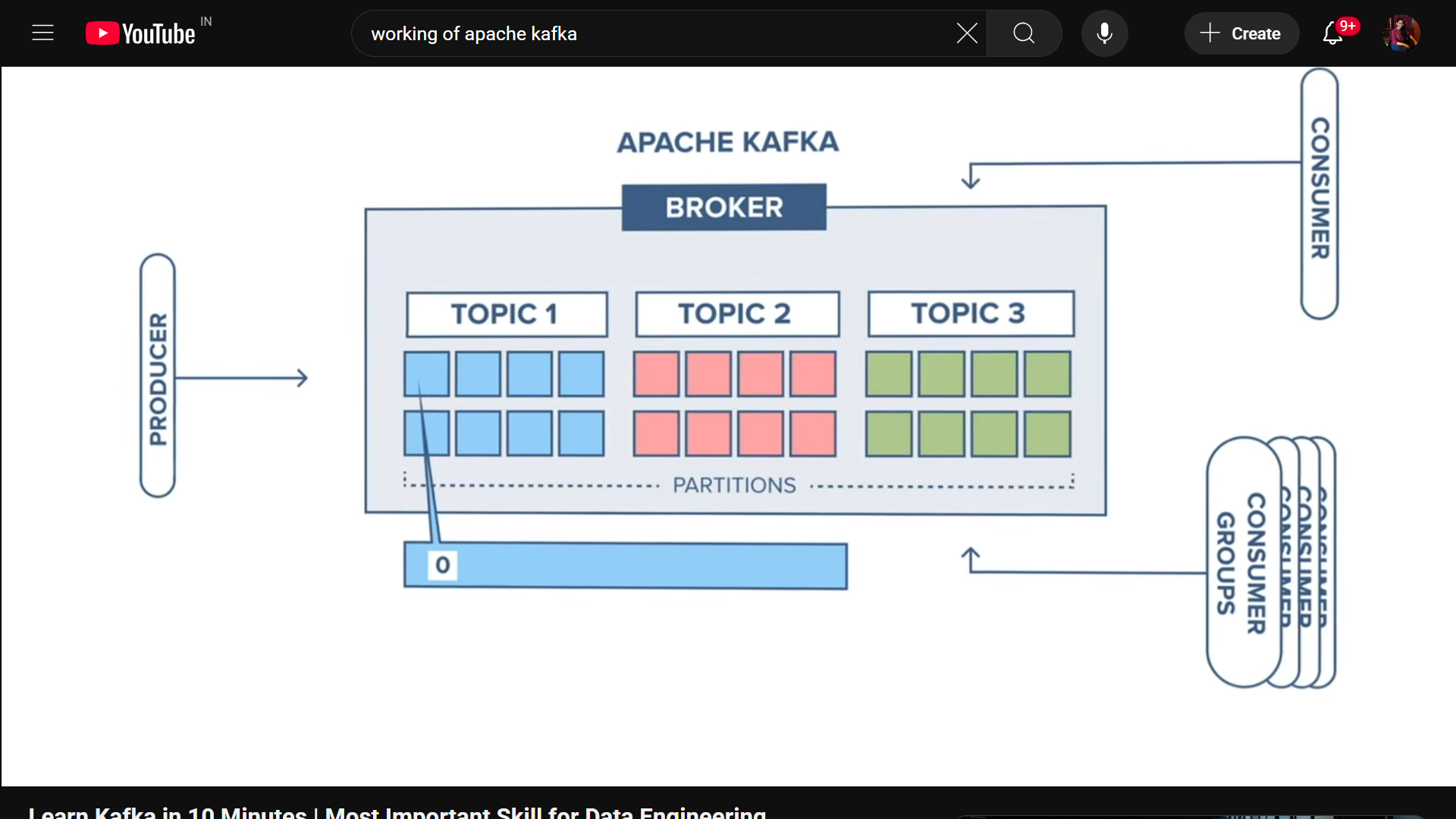
**Apache Kafka is a tool used to send and receive messages between different parts of an application.**  
It helps services talk to each other without being directly connected.  
Kafka is fast, handles a large amount of data, and is often used when we need real-time data processing — like in chat apps, online orders, or tracking user activity.

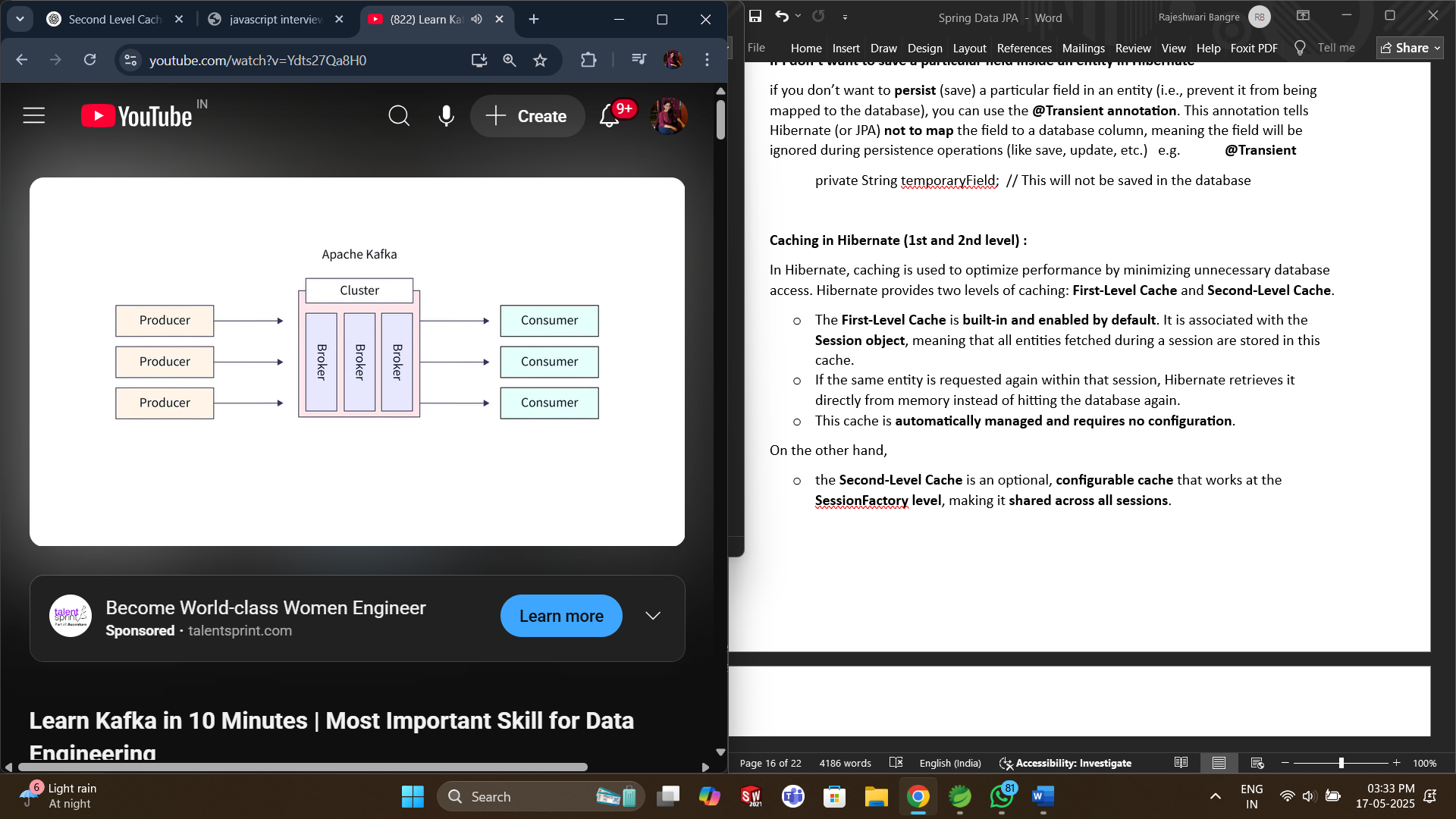
Kafka is built by **LinkedIn** and later open-sourced under the **Apache Software Foundation**. Kafka has high throughput (Operation Per Sec) but low storage space so it is used with database (PostgresSQL) to increase throughput i.e. bulk of data can be stored at once consuming less time.

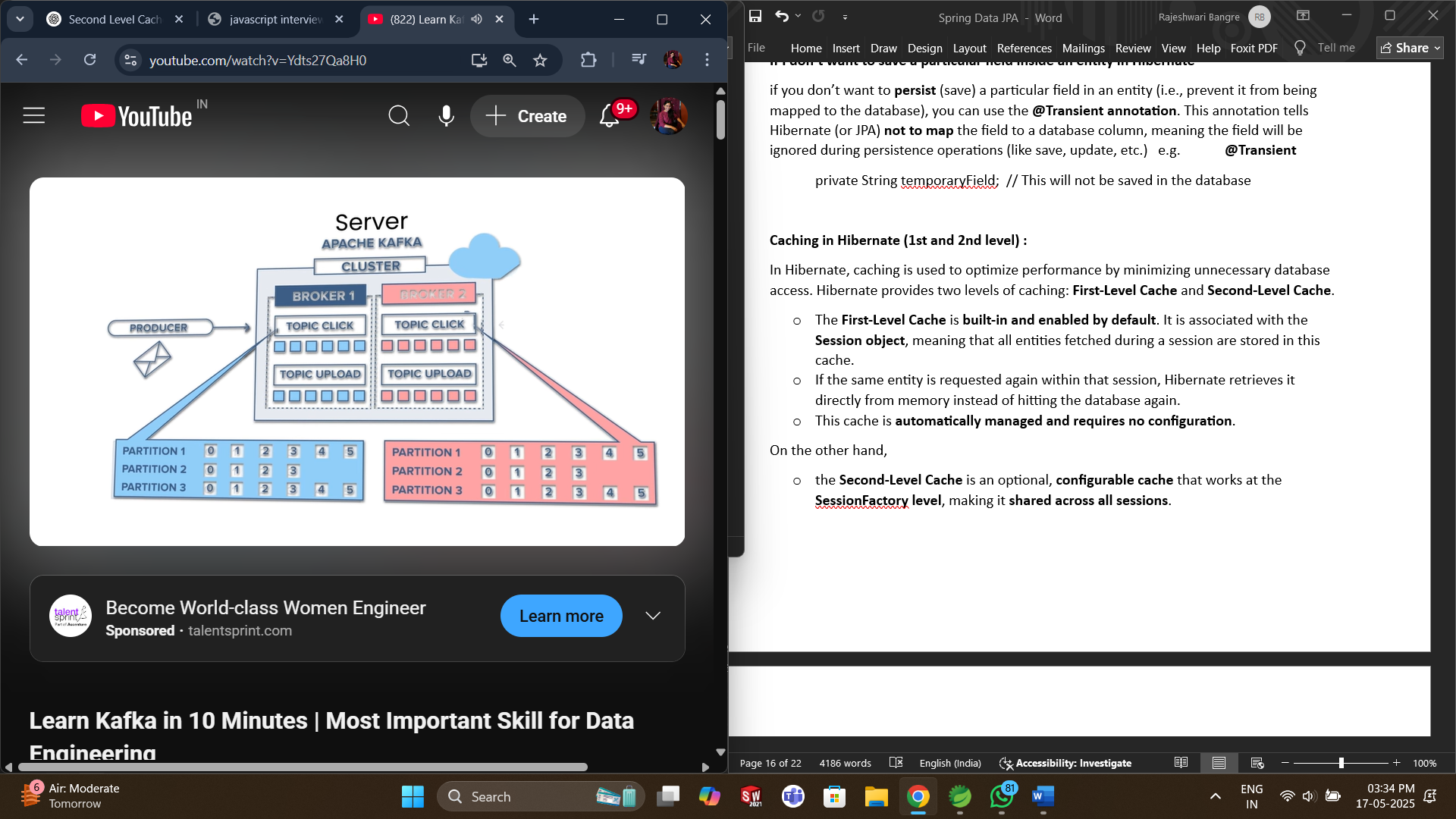
**Internal** : Kafka (9092 default port) has producer, consumer, topic, partition. **Kafka is managed by Zookeeper** (2181 default port)

* **Producer –** produce message
* **Consumer –** consumes message
* **Topic** – Kafka server is divided into categories – under which topic u want to save data
* **Partition** – topic is divided into partition based on meaningful parameters so that we don’t cluster everything under a topic. Also adding data must be consistent here e.g. we can divide the rider service on Zomato location wise but dividing it into time slots will be foolishness. Partitioning works on indexing e.g. Partition 0, Partition 1









Kafka **auto balances** the data between consumers e.g. if there are 4 partition and 4 consumers then each partition will provide data to each consumer instead of overloading to 1 consumer.

Rule says that 1 consumer can consume multiple partition but 1 partition can have only 1 consumer

Group consumer – Consumer can be grouped. There will be auto balancing on group level.

We can run Kafka as queue / pub-sub model

* Queue Architecture : 1 Producer 1 consumer (one – one mapping)
* Pub-sub Architecture : 1 Producer multiple consumer (one – many mapping)



**gRPC**

gRPC **(Remote Procedural call)** is an open-source, high-performance RPC framework developed by Google. It allows communication between distributed services and system, often used in microservices architecture.

* gRPC uses **HTTP/2** for transport, it is faster and supports multiplexing.
* It uses **Protocol Buffers** for serialization structured data – smaller and faster than JSON and XML.
* Supports all types of communication
* Supports all languages – Java, Python, Go, C++ etc.

Used in mobile to server networks, communication between microservices.

**Persistent & Real-Time Communication**

**1. WebSocket**

**WebSocket is a communication protocol that provides full-duplex (two-way) communication between client and server over a single, long-lived TCP connection.**  
Unlike HTTP, where the client must initiate every request, **WebSocket allows both the client and server to send messages at any time** — making it ideal for real-time applications.

**Eureka Server :**



**Core Concept**

Eureka acts as a **service discovery mechanism**. Microservices register themselves with Eureka, and other services can discover them by querying the registry.

**Eureka Architecture**

Components:

1. **Eureka Server** – Central registry where all services register.
2. **Eureka Client** – Microservices that register themselves with the Eureka server and can discover other services.

**How Eureka Works – Step by Step**

1. **Service Registration (Client to Eureka)**

When a microservice (Eureka Client) starts, it sends a registration request to the Eureka Server.

This includes metadata like:

1. Service ID
2. Host and Port
3. Health Check URL
4. Home Page URL

Eureka Server stores this information in its registry.

1. **Heartbeat Mechanism**

Eureka Clients send a heartbeat every 30 seconds (default) to let Eureka Server know they are alive.

If the server doesn’t receive a heartbeat in 90 seconds (default), it assumes the service is down and removes it from the registry.

1. **Service Discovery (Client to Eureka)**

When a microservice wants to call another service, it queries the Eureka Server using the service ID.

Eureka returns a list of all available instances for that service.

The calling service chooses one (typically using **Ribbon load balancer, Round Robin** or some custom strategy).

**Inter Service communication in Microservice :**

**Client** → **API Gateway** → **Order Service** → **User Service (via Feign)**

↑

**Eureka Server**

**Setup :**

1. **Eureka Server**

Dependency – eureka server

Annotation – in main class, use @EnableEurekaServer

1. **User Service (Registers with Eureka)**

Dependency – eureka client

Annotation – in controller class, use @restController, @RequestMapping(“/users”), get API

1. **Order Service (Uses Feign to Call User Service)**

Dependency – Feign Client, eureka client

Annotation – in main class, use @EnableFeignClients, @FeignClient(name = "user-service")

File – create feign client class and declare a get API to get something from User

E.g. @FeignClient(name = "user-service") // Registered service name

public interface UserClient {

@GetMapping("/users/{id}")

String getUser(@PathVariable("id") String id);

}

Define controller class for same get API

1. **API Gateway**

Dependency – gateway, eureka client

**API Gateway** is the entry point, Routes external requests to appropriate microservice

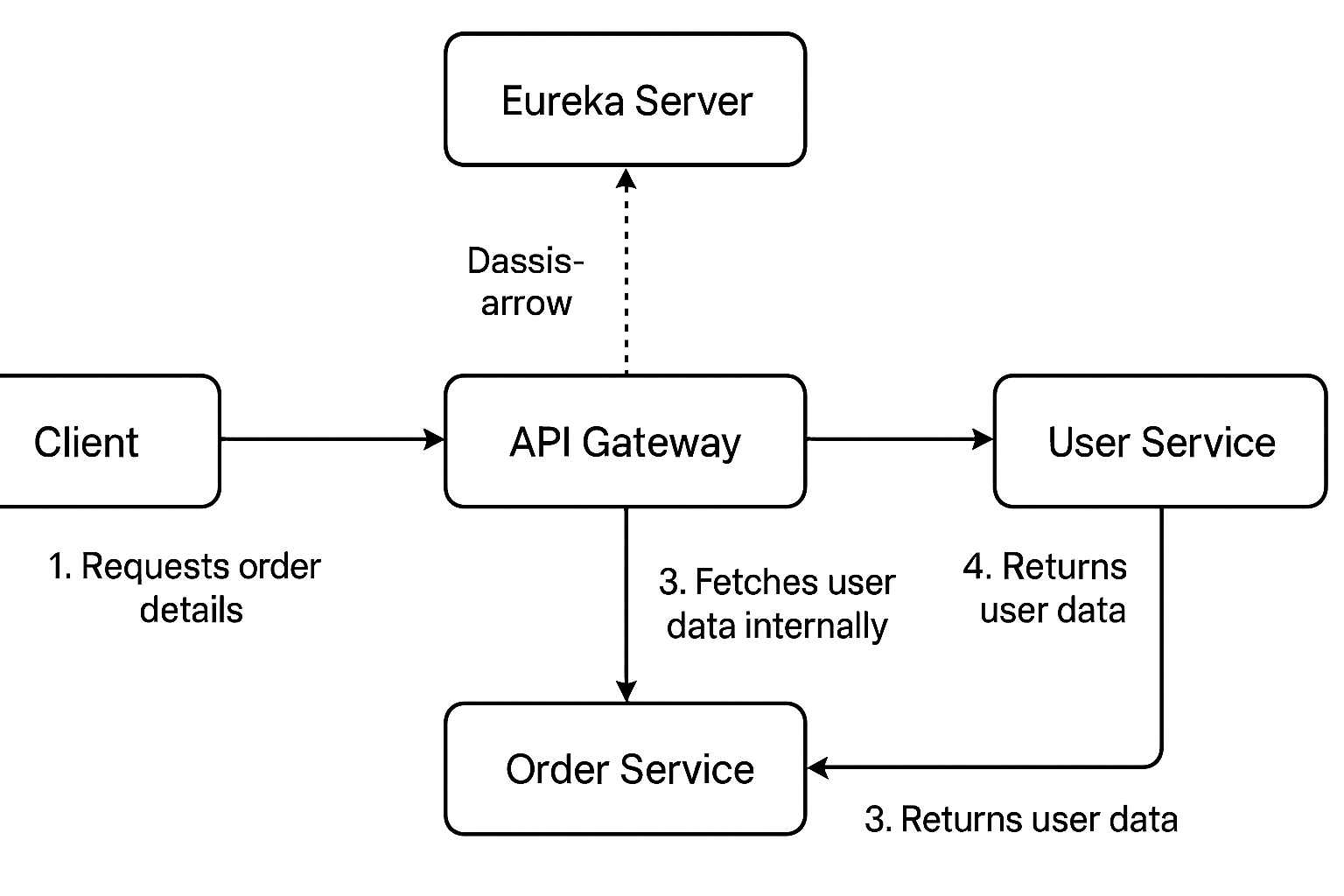
**Order Service** serves order details, Main logic; fetches users via Feign and returns result

**User Service** serves user details

**Feign Client** is used by Order Service to call User Service**, Enables internal service-to-service communication**

**Eureka Server** registers and discovers services (knows where each service lives).

**Step-by-Step Communication Flow**

****

**Step 1: User Sends HTTP Request**

* This hits the **API Gateway** (port 8080)
* API Gateway is configured to forward /orders/\*\* to the **Order Service**

**Step 2: API Gateway Routes the Request**

* Gateway uses service discovery to find the service named order-service from Eureka
* It forwards the request to the actual endpoint:

<http://order-service/orders/101>

(using **LoadBalancer** behind the scenes)

**Step 3: Order Service Processes the Request**

* OrderController receives /orders/101
* While preparing the order response, it needs user details (e.g., user who placed the order)

**Step 4: Order Service Calls User Service via Feign**

* It uses the User client and calls the get API to get the Users
* It asks the Eureka server for user-service port
* Eureka responds with the registered IP and port
* Then Feign client uses that port and sends the request to get Users

**Step 5: User Service Returns the User Data**

* UserController in user-service receives the call

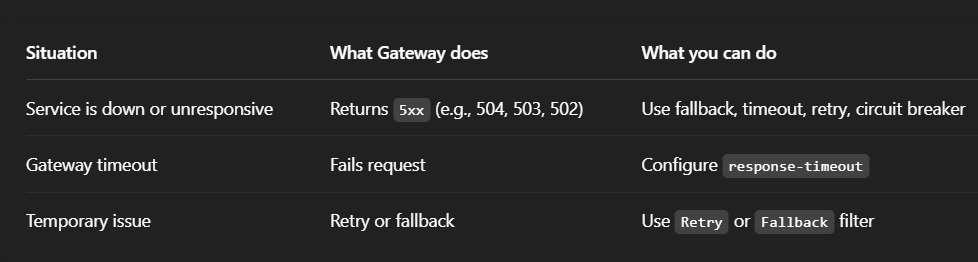
**Step 6: Order Service Builds Final Response**

* OrderService receives the user data
* Combines it with order info:

**Step 7: API Gateway Returns Final Response to User**

* The response travels back from: Order Service → API Gateway → Client
* User sees the combined response on browser or frontend
* If API Gateway fails to get the response it timeout, retries and use circuit breaker still if it fails then in order to show the meaningful response it uses fallback() method
* In Spring Cloud Gateway, this can be done using:

Resilience4j or Hystrix (older) for fallback



**Circuit Breaker :**

The **Circuit Breaker** is a **design pattern** used in **microservices and distributed systems** to **prevent a failure in one service** from crashing or overloading the entire system.

Circuit Breaker helps isolate failures and prevents the system from making repeated calls to a failing service. It works in **three states (Closed, Open, Half-Open)** and recovers automatically once the service is healthy again.

E.g. Imagine a fuse in an electrical circuit — if there's a problem or overload, the fuse **breaks the circuit** to prevent damage. Similarly, the circuit breaker in software **stops calls to a failing service temporarily**.

**States of a Circuit Breaker:**

1. **Closed (Normal)**

Circuit is closed when everything is working fine and request flow is normal

1. **Open (Tripped)**

When too many failures are detected then circuit opens and future requests are blocked temporarily.

1. **Half-Open (Test Mode)**
   * After a timeout, the system sends **a few test requests**.
   * If successful, it goes back to **Closed**.
   * If failed again, it goes back to **Open**.

There is an API gateway and eureka server and microservices.



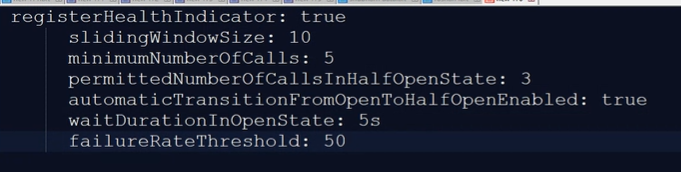
**Advantages :**

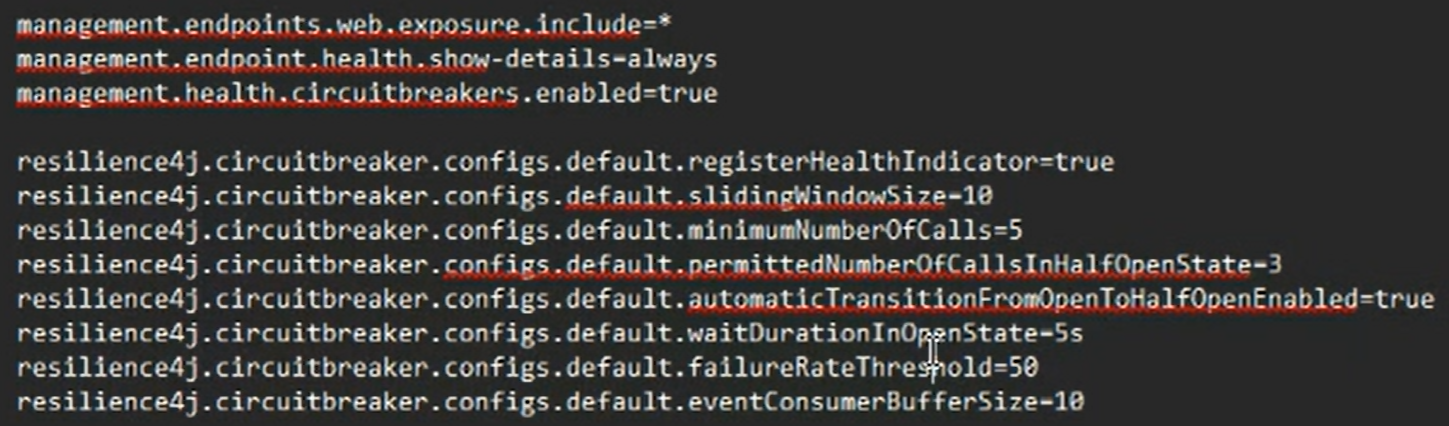
* Prevent cascading failures
* Protect healthy parts of the system
* Improve system resilience and stability
* Reduce response time during service downtime

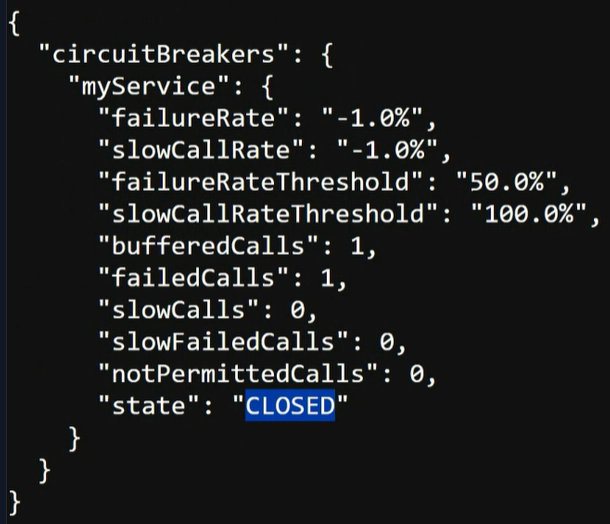
**Tools / Libraries**

* **Resilience4j** (modern and lightweight)
* **Hystrix** (Netflix, now in maintenance mode)
* **Spring Cloud Circuit Breaker** (integrates with Resilience4j or Hystrix)

Sliding window:







**Circuit breaker status with the help of actuator**

**Various ways to implement microservices :**

There are different ways to implement microservices like AWS, Microsoft Azure they provide infrastructure to implement microservices. But I know about the spring cloud based microservices because spring is also providing a library to implement microservices architecture mainly it provides spring cloud. So, in spring cloud we can have API Gateway, Eureka Server.

Further explain about Gateway, server.

**There are several ways to implement microservices architecture, and the choice depends on factors like scalability, team expertise, time-to-market, and operational overhead. Let me break down the main approaches briefly:"**

1. **Container-Based Microservices** –
2. **Serverless Microservices** –
3. **Managed Cloud Services (PaaS)** –
4. **Traditional VM-Based Approach** –
5. **Event-Driven Architecture** –
6. **Service Mesh** –
7. **API Gateway-Based Architecture** –

**"In a real project, I might combine several of these—like using containers for core services, serverless for background jobs, and an API Gateway to tie them together."**

**Spring Cloud Config :**

**In a microservices architecture, managing configurations centrally and securely is critical — especially as the number of services grows. To handle this, I follow several strategies depending on the project's scale and requirements.**

**Spring Cloud Config** provides **centralized external configuration** for microservices.

Instead of hardcoding configs (e.g., DB URL, API keys, feature flags) in each service or keeping them in multiple application.properties files, you store them in a **central config repository**, usually a **Git repo**.

**What it provides :**

**Centralized configuration**: One place to manage properties for all services.

**Dynamic updates**: Change configs **without redeploying** services.

**Environment-specific configs**: Automatically load different configs for dev, test, prod.

**Secure and version-controlled**: Git integration provides rollback and audit.

**How it works:**

1. **Spring Cloud Config Server** – Reads config files from Git/Filesystem/HashiCorp Vault.
2. **Spring Cloud Config Client** – Microservices fetch their config from the server at startup (or via /actuator/refresh if Spring Cloud Bus is used).

**Spring Boot Actuator :**

**Spring Boot Actuator** provides **production-ready endpoints** to monitor and manage applications.

**Health checks** for load balancers & Kubernetes probes

**Monitoring** with Prometheus, Grafana, or AWS CloudWatch

**Debugging** and system insights

**Dynamic config refresh** via Spring Cloud Bus + /refresh

It exposes REST endpoints like:

* /actuator/health – Check if the service is up
* /actuator/info – Metadata like app version
* /actuator/metrics – Performance metrics (memory, CPU, DB, etc.)
* /actuator/env – View environment properties
* /actuator/beans – See all Spring beans
* /actuator/refresh – Reload configs dynamically (used with Spring Cloud Config)

**How to add Actuator to your project**

Step 1. Add dependency in pom.xml

Step 2. Add application.properties – management.endpoints.web.exposure.include = \*

Step 3. Run your Spring Boot app Actuator will expose endpoints automatically

**Saga Design Pattern :**

In microservices, a single business operation may span multiple services (e.g. order service, payment service, inventory service).

But traditional database transactions (ACID) don’t work well across microservices – they’re hard to scale and tightly coupled.

So, the Saga patten replaces distributed transactions with a series of local transactions that are coordinated using either Choreography (event-based), Orchestration (command-based)

In **Choreography**, there is no central controller — each service performs its task and publishes an event; other services listen to these events and respond accordingly. It’s **simple** and **loosely coupled** but can become **hard to manage** as the system grows.

In **Orchestration**, a central **Saga orchestrator** directs the flow by invoking each service in a defined sequence and handling failures with compensating actions. This approach offers **better control** and **visibility** but introduces **some coupling** to the orchestrator.

**How Microservices are configured in your project?**

Typical configuration in a Spring Boot microservices project:

Service Discovery: Using Eureka Server

API Gateway: Using Spring Cloud Gateway

Inter-service Communication: Using FeignClient or RestTemplate

Configuration Management: Using Spring Cloud Config Server

Containerization: Using Docker

Load Balancing: Using Spring Cloud LoadBalancer

Fault Tolerance: Using Resilience4j Circuit Breaker

**Questions :**

**What will you do if any service is down?**

I will follow a systematic procedure first monitor the application, check logs through Spring Boot Actuator or any other that we are using like Prometheus, Grafana etc

* Verifying the problem by reproducing the bug, check dependencies.
* If critical, then for temporary purpose use Circuit Breaker, fallback methods to avoid crashing or overloading.
* Restart if it’s a crash and rollback recent deployment.
* Communicate issue with team via incident managing platform for example Jira.
* Perform RCA (Root Cause Analysis)

**What if System or anything crashes in your application?**

I will follow approach that’s require

First detach the crash and monitor them for example Actuator.

Identify the root cause by checking logs.

**How do two services communicate?**

**What is Application-Oriented Programming?**

**What is the difference between monolithic and microservice architectures?**

**How do microservices communicate with each other in your project?**

**Explain how Eureka, FeignClient, and Ribbon load balancer work together in a Spring Cloud setup.**

**How do you handle failure between microservices (timeouts, fallbacks)?**

**What strategies do you use for configuration management in microservices?**